DTIC
ELECTE
FEB 0 3 1994
S F D

Exploiting Symmetry in the Model Checking of Relational Specifications

Daniel Jackson

December 1994
CMU-CS-94-219

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Errors in a software design can be detected early on by analyzing a formal model expressed in a specification language such as Z. Since software designs tend to involve infinite (or at least very big) state spaces, it has been assumed that this analysis cannot be automated. Consequently, few formal specifications have been extensively analyzed, and the potential for early detection of errors has not been realized.

This paper argues that, while proving properties of designs may be intractable, detecting errors may not be. State transitions of an operation can be enumerated exhaustively, within a 'scope' defined by the user that places a bound on the size of state components. Symmetry can then be exploited to reduce this finite state space. A state can be shown to be symmetrical, in the context of the analysis, to a state already examined, and thus guaranteed not to reveal an error.

Preliminary experiments with a prototype are promising. A small scope often seems sufficient to catch errors, and exhibits enough symmetry to reduce search by a factor of 10 or more.

This research was sponsored in part by a Research Initiation Award from the National Science Foundation (NSF), under grant CCR-9308726, by a grant from the TRW Corporation, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA), under grant F33615-93-1-1330. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either express or implied, of ARPA, NSF, TRW or the United States Government.

This document has been approved for public release and sale; its distribution is unlimited.

19950201 007

Keywords: Symmetry, model checking, software design, formal specification, Z, relational calculus.

1 *Introduction*

The remarkable success of model checking in analyzing hardware designs raises an obvious question. Can the same technique – namely massive enumeration of cases – be applied to software designs? The payoff would be spectacular.

Pioneers of formal specification realized that its primary justification was not formalism per se (since, after all, judicious use of informal mathematics can provide much of the intellectual benefit with fewer restrictions), but the opportunities it opened up for mechanical analysis. Questions about the consequences of design decisions might be cast as theorems and then answered automatically [GH80].

Unfortunately, such theorems tend to be undecidable. Software, unlike hardware, is often designed initially with the assumption of unlimited resources, and, unless these can be modelled purely as sets, no interesting property can be determined effectively. And even if the state space of the software system is finite, it is usually so large that any kind of naive enumeration is hopeless. A simple binary relation over a set of three elements can have 2^9 values; take three such relations and the size of the state space already matches that of a complex hardware design. For these reasons, most research on analysis of specifications has concentrated on theorem proving, and the possibility of automatic analysis by model checking has been dismissed as unrealistic: an idea too stupid to be seriously pursued.

Checking software designs by enumerating states is not, however, as stupid as it may appear, for three reasons. First, finding counterexamples may be feasible when proof is not. Complete search of an infinite state space is impossible, of course, so model checking will not, in general, be capable of demonstrating that a design has a certain required property. In practice, however, most designs contain errors. In economic terms, early detection of errors has more value than assurance of correctness, which, as detractors of formal methods have repeatedly observed, can never be guaranteed anyway. A formal specification is by nature an abstraction and by necessity partial, so that the distinction between high confidence of a design's correctness and perfect assurance cannot be maintained. A model checker that examined only a tiny proportion of the state space but found a high propertion of errors in the design would be very useful.

Second, the transition relation of a software design is usually complex enough to merit 'single-step' analysis: one often wants to determine properties of a single execution of some operation. The behaviour of hardware, on the other hand, is usually viewed over sequences of simple steps. This kind of temporal analysis requires, at the very least, a record of which states have been visited; the size of this data struc-

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification ....
By ....
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

ture strictly limits the number of states that can be examined. For a single-step analysis, the states can be generated in order and discarded.

Third, it may be possible to find ways to search enormous state spaces. Witness the discovery in hardware verification that a small binary decision diagram can characterize a huge set of states, allowing vastly bigger designs to be checked [BC+90]. And abstraction can sometimes enable even an infinite space to be searched by partitioning it into a finite number of equivalence classes [CGL92, Jac94].

This paper describes a model checking algorithm for analyzing designs written in a minimal specification language intended to capture the essence of Z [Spi89]. Z was chosen for its popularity, because it is model-based (and thus more overtly suited to model checking than algebraic languages such as Larch), and because it has a simple set-theoretic semantics (unlike VDM, which adopts the domains of denotational semantics). The purpose of the algorithm is to detect counterexamples, so the search is confined within a 'scope' that limits the size of the objects that model the state. A single-step property of a Z specification amounts to a logical assertion, typically an implication whose hypothesis is the specification of an operation and whose consequence is a property whose truth is to be determined. A counterexample is thus a binding of variables (the state components) to values (particular sets, relations, etc.) for which this assertion is false.

The notable feature of this algorithm is that it exploits symmetry in the state space. Some states can be eliminated from the search a priori, since they are known to be equivalent, in the context of the claim being evaluated, to states that have already been examined. This is determined fully automatically and, unlike techniques based on abstraction, requires no hints from the user. Preliminary experiments indicate that exploiting symmetry usually reduces the search by a factor of 10 or more.

2 *Some Examples (and Counterexamples)*

Consider a toy telephone system whose state consists of two components:

$Called: Ph \leftrightarrow Num$
$Net: Num \rightarrow Ph$

The first is a relation that associates with each phone zero, one or more numbers with which it has a call in progress. The second is a function mapping numbers to phones. The set of active connections between phones is a relation

$Conns: Ph \leftrightarrow Ph$
$Conns = Called \circ Net$

obtained by composing the state components as relations. The actions performed by users at their phones can be represented as operations on the state of the system. To add a party with number n to a conference call from phone p, for example, one might execute

$op\ bridge\ (p{:}\ Ph,\ n{:}\ Num)$
$\quad pre\quad p \in dom\ Called \wedge n \notin ran\ Called$
$\quad post\ \ Called' = Called \cup \{(p,n)\} \wedge Net' = Net$

The precondition says when the operation will succeed: the phone p must already have a call in progress, and no other phone must be calling the number n. Some other part of the specification of the system will say what happens when the precondition is violated; including it here allows us (temporarily at least) to ignore that complication. The postcondition gives the effect of the operation: the phone p is mapped to n in *Called*, but *Net* is unchanged.

(This specification incidentally, is not in Z. The unusual role of invariants and preconditions in Z is, from the point of view of model checking, an irrelevant distraction; what matters is the representation of the state in terms of relations and the description of an operation's behaviour with a logical formula. Also, the symbol $\circ$ denotes backward not forward composition in Z, contrary to the standard usage adopted here.)

Many useful properties of an operation can be expressed as invariants on the state. Suppose, for example, that for billing purposes we forbid a phone from being both a caller and a callee at once:

$$invB \equiv (dom\ Conns \cap ran\ Conns = \varnothing)$$

To see whether the operation is consistent with this assumption, we can check the formula

$$invB \wedge pre \wedge post \Rightarrow invB'$$

where *invB′* is the invariant *invB* with *Conns′* substituted for *Conns*. A counterexample to this formula would yield a state in which the invariant holds and the operation can execute successfully but result in a state that violates the invariant.

The checker finds such a counterexample after examining only 38 states:

$$Called = \{(p_1, n_2)\}$$

$Net = \{(n_1, p_1)\}$
$p = p_1$
$n = n_1$

This counterexample reminds us that we forgot the obvious: *n* should not refer to a phone already making a call. But its small size also draws our attention to two more subtle points. First, we neglected the possibility that a phone call itself, which we shall surely want to rule out. Second, we never required that *Net* be total, or that calls only be made to numbers attached to phones.

Here's a second example. The precondition included

$n \notin ran\ Called$

with the intent of ensuring that we not connect to a phone that has already been called by another phone. Formulating this as an invariant, we might say that two different phones cannot call the same phone

$invC \equiv (p, p'') \in Conns \wedge (p', p'') \in Conns \Rightarrow p = p'$

or equivalently that the inverse of *Conns* is a function

$invC \equiv Conns^{\sim} \in (Ph \rightarrow Ph)$

This invariant is also not preserved, and after checking 185 states, the checker finds

$Called = \{(p_1, n_1), (p_2, n_2)\}$
$Net = \{(n_2, p_3), (n_3, p_3)\}$
$p = p_1$
$n = n_3$

The essence of this counterexample is the value of *Net*. A phone may be called through one of several numbers. To check if a number corresponds to a phone that has been called it is therefore not sufficient to determine that the number itself was not called. We must either require that *Net* be injective or strengthen the precondition.

This example, although far from a real specification of a telephone switch [MZ94], neverthess illustrates some realistic features of counterexample generation. First, we have seen that quite subtle errors can be revealed in small counterexamples; in all three cases, considering only 3 phones and 3 numbers was enough. Second, a counterexample has an immediate, tangible benefit. It can be hard to determine, when a theorem prover fails to prove a theorem, what has gone wrong; a counterexample leaves little room for doubt, and often reveals several faults at once.

The remainder of the paper explains how the search for a counterexample can be drastically reduced by exploiting symmetry in the state space. A rough idea of how this works can be gained by looking at some of the states that arise in our example.

Figure 1 shows a state just before execution of the *bridge* operation. The two graphs depict the relations *Called* and *Net*; the circles show the values of p and n, written over the nodes of *Called* to show where the new pair will appear in *Called′*. The search is being conducted within a scope of three phones and three numbers.

Suppose we apply a permutation to the entire state that exchanges n_1 and n_2, leaving *Called* unchanged by altering *Net*. The permutation is therefore a symmetry of *Called* but not of *Net*. It is also, however, a symmetry of the entire state, in the following sense. The logical formula being checked (in the case of either invariant discussed above) is cast entirely in terms of *Conns*, the product of *Called* and *Net*, and *Conns′*, the product of *Called′* and *Net*. Permuting n_1 and n_2 changes *Net*, but neither of these products, both of which include the pair (p_1, p_1) regardless of whether the intermediate element is n_1 or n_2. Having checked the state shown, the symmetrical state with the value of *Net* altered (the edge moving as shown to the new dotted position) can therefore be safely omitted from the search.

Symmetry thus allows fewer states to be examined before discovering a counterexample. Table 1 shows its effect in reducing the state space. The checker was set to ignore counterexamples and complete the search; the total number of states searched is shown for different declarations of the *Net* relation, followed by the reduction factor. The worst case is when *Net* is injective, because it eliminates

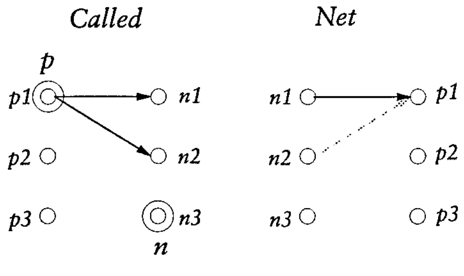

Figure 1: An example of a symmetry

the symmetry of two domain elements that map to the same range element.

The first set of results applies to the checking of the firstof the invariants discussed above, and the second set applies to the second invariant. There is more symmetry to exploit in the second invariant, because it does not relate the domain and range of *Conns*, which may thus be permuted without visible effect.

Type of *Net*	Checking *invB*		Checking *invC*	
	#States	Factor	#States	Factor
total function	11,696	11	2002	62
injective function	16,866	5	3214	27
partial function	22,966	13	4606	64
arbitrary relation	168,984	14	34,492	68

Table 1: Reduction of the state space due to symmetry

3 Relational Calculus

The simplicity of Z, and to a great extent its power and flexibility too, derives from treating functions, mappings and sequences uniformly as sets of pairs – that is, relations. Rather than describing the checking of Z directly, we shall instead consider formulae in a much simpler relational language. This exposes the fundamental issues by stripping away syntactic niceties and makes the algorithm's application to similar languages (such as AMN) more apparent. The language may also turn out to be a good intermediate language into which Z can be translated prior to checking. It is based on the relational calculus given its current form by Tarski and others (and nicely summarized in Chapter 2 of [SS93]).

The language is purely relational: there are no sets or atoms. A relation is binary, but may be heterogeneous and thus has a type (*S*,*T*). When we give a model later, we will interpret *S* and *T* as the sets from which the domain and range elements are drawn respectively. Basic types such as *S* and *T* are distinct; there is no subtyping. Nor may the elements of *S* and *T* themselves be relations: the language is strictly first-order. We shall assume that a formula is coupled with a type declaration that assigns a type to each relation variable, and that the formula itself is well-typed.

The terms of the language are made of relation variables *P*, *Q*, *R*, etc., and from subterms *s*, *t*, etc., composed with

- union $s \cup t$ and intersection $s \cap t$: these correspond to the set union and intersection of the relations represented as sets of pairs;

- composition $s \circ t$, also called relational product;
- complement s', defined with respect to the type of s, giving the relation of the same type whose pairs are all those not in s;
- inverse $s^{\sim}$.

There are also three constants: 0 (the empty relation), L (the universal relation) and I (the identity). To be precise, we should distinguish instances of these constants that have different types, but it is easy to infer the types from the context, so we shall not bother. In the formula

$$P = Q \circ L$$

for example, if P has type (S, T) and Q has type (S, U), then the appropriate universal relation L has type (U, T), and is interpreted as the cartesian product of the sets denote by U and T.

Applying the ordering $\subseteq$ to terms is the only way to generate elementary formulae; $s \subseteq t$ will be true when the set of pairs denoted by s is a subset of the set denoted by t. The ordering is antisymmetric, so $s = t$ when $s \subseteq t$ and $t \subseteq s$. Finally, formulae may be joined by the logical connectives, $\wedge$, $\vee$, and $\neg$.

To see how to express familiar specifications in this language, note that a subset u of a set X can be represented as a relation on $X \times Y$ that pairs each element of u with every element of Y. A relation u thus denotes a set if

$$u \circ L = u.$$

Similarly, a point x in X is a relation on $X \times Y$ that pairs the single element x with every element of Y. A relation x is a point if

$$x = x \circ L \ \wedge \ x \circ x^{\sim} \subseteq I \ \wedge \ \neg x = 0.$$

A pair of points (x, y) is represented by the relation

$$x \circ y^{\sim}$$

It is easy to write formulae that classify relations in the manner of Z's type declarations. A relation R is total if

$$L = R \circ L$$

and a function if

$$R \circ R^{\sim} \subseteq I$$

It is surjective if its inverse is total, injective if its inverse is a function, and bijective if it is injective and surjective.

The familiar operators of Z are also easily defined. The domain of a relation R is just $R \circ L$, and its range is $R^{\sim} \circ L$. The domain restriction of a relation R to a set S, written in Z as $S \lhd R$, is $S \cap R$. Relational override is defined by

$$P \oplus Q = (P \cap (Q \circ L)') \cup Q.$$

4 A Model of the Relational Calculus

Reasoning in the relational calculus can be conducted purely algebraically, without appealing to any concrete representation of relations. Since our purpose, however, is finding counterexamples – concrete values of variables for which a formula does not hold – the representation of relations is critical.

We shall start with a conventional model of relations. The meaning of a formula f and its constituent subformulae and terms is defined with respect to an interpretation $\langle U, \tau, \rho \rangle$ consisting of

- a universe U of values,
- a mapping τ that associates with each type name T in the type declarations of f a set $\tau[\![T]\!] \subseteq U$, disjoint from the set $\tau[\![S]\!]$ associated with a different type name S, and
- a mapping ρ that associates with each variable R of type (S, T) in f a set of pairs $\rho[\![R]\!] \subseteq \tau[\![S]\!] \times \tau[\![T]\!]$.

The meaning of a formula is defined inductively over the structure of the formula. Consider terms first. The meaning of a relation variable is given directly by the interpretation

$$M[\![R]\!] = \rho[\![R]\!]$$

For the constants, the zero relation 0, the identity relation I_T of type (T, T) and the universal relation L_{ST} of type (S, T), we have

$$M[\![0]\!] = \varnothing$$
$$M[\![I_T]\!] = \{(x, x) \mid x \in \tau[\![T]\!]\}$$
$$M[\![L_{ST}]\!] = \tau[\![S]\!] \times \tau[\![T]\!]$$

The complement s' of a term s of type (S, T) denotes the largest relation of the same type that includes none of the pairs in s

$$M[\![s']\!] = \{(x, y) \in \tau[\![S]\!] \times \tau[\![T]\!] \mid (x, y) \notin [\![s]\!]\}$$

The other relational operators have their conventional meaning:

$$M[\![s \cup t]\!] = \{(x, y) \mid (x, y) \in M[\![s]\!] \ \mathit{or}\ (x, y) \in M[\![t]\!]\}$$

$M[\![s \cap t]\!] = \{(x, y) \mid (x, y) \in M[\![s]\!] \text{ and } (x, y) \in M[\![t]\!]\}$
$M[\![s \circ t]\!] = \{(x, y) \mid \exists z.\ (x, z) \in M[\![s]\!] \text{ and } (z, y) \in M[\![t]\!]\}$
$M[\![s^{\sim}]\!] = \{(y, x) \mid (x, y) \in M[\![s]\!]\}.$

The ordering on relations denotes the subset ordering on their pair sets:

$$M[\![s \subseteq t]\!] = \forall (x, y) \in M[\![s]\!].\ (x, y) \in M[\![t]\!]$$

And finally, the logical connectives:

$M[\![f \wedge g]\!] = M[\![f]\!] \text{ and } M[\![g]\!]$
$M[\![f \vee g]\!] = M[\![f]\!] \text{ or } M[\![g]\!]$
$M[\![\neg f]\!] = \text{not } M[\![f]\!]$

Under a given an interpretation, the meaning of a formula is either true or false. If it is true, the interpretation is said to be a *model* of the formula; if it is false, the interpretation is a *counterexample*. A formula that has a model is *satisfiable*; a formula that has no counterexamples is *valid*.

5 A Different Model

Consider checking the formula

$$\neg P \subseteq Q \ \vee \ Q \circ R \subseteq \ P \circ R$$

which, were P and Q to be exchanged in one of the subformulae, would express the monotonicity of composition, but as it stands is not valid. One counterexample maps P to the empty relation, Q to the singleton $\{(a, b)\}$ and R to $\{(b, c)\}$. Mapping R instead to $\{(a, b)\}$ does not give a counterexample.

Suppose we are enumerating interpretations in a search for counterexamples, and we generate this latter interpretation first. Generating values for R, we might then consider $\{(a, c)\}$. This is guaranteed not to be a counterexample for a simple reason: the second element of the pair is irrelevant, and cannot affect the meaning of the second subformula. In general, any permutation of the range elements of R will preserve the meaning of the formula. What matters here is the matching of the ranges P and Q to the domain of R; the counterexample that maps R to $\{(b, c)\}$ works because b matches the second element of the pair in Q but no second element in R, and thus distinguishes Q and R.

This suggests that to exploit such symmetries we should focus not on the values of the elements in relations' pairs, but in the mappings between them. We therefore define a new kind of interpretation. First, we must distinguish the occurrence of an element in different *places*: the range elements of R in the example above must be disjoint from the domain or range elements of P or Q so that we can refer to them independently. To do this, we label each occurrence of a type name T in the type declarations with an index to distinguish it from other occurrences of the same name. For example the declarations

$P: (S, S)$
$Q: (T, U)$
$R: (S, T)$

would be transformed to

$P: (S_1, S_2)$
$Q: (T_1, U_1)$
$R: (S_3, T_2)$

A *wired interpretation* $\langle U, \tau, \rho, W\rangle$ of a formula f whose type declarations have been transformed in this manner consists of:

- a universe U of values, as before,
- a mapping τ that associates with each type name T_i in the type declarations of f a set $\tau[\![T_i]\!] \subseteq U$. These sets must, as before, be disjoint; furthermore, sets $\tau[\![T_i]\!]$ and $\tau[\![T_j]\!]$ associated with two type names that differ only in their index must have the same cardinality,
- a mapping ρ that associates with each variable R of type (S, T) in f a set of pairs $\rho[\![R]\!] \subseteq \tau[\![S]\!] \times \tau[\![T]\!]$, as before, and
- an additional component W.

W is called the *wiring*. It is an equivalence relation on U with the following property:

- for each x in $\tau[\![T_i]\!]$ there is exactly one y in each set $\tau[\![T_j]\!]$, and no z in any $\tau[\![S_k]\!]$, such that xWy

where S and T are distinct type names. Intuitively, W matches elements in different places that have the same value in the underlying type.

Operators on a single relation, and the logical connectives, are interpreted just as before. The meaning of a term involving two relations, however, now depends on the wiring; where equality appeared in the definition before, the wiring equivalence W now takes its place.

$$M[\![s \cup t]\!] = \{(x, y) \mid (x, y) \in M[\![s]\!] \textit{ or } \exists (x', y') \in M[\![t]\!].\ xWx' \textit{ and } yWy'\}$$
$$M[\![s \cap t]\!] = \{(x, y) \mid (x, y) \in M[\![s]\!] \textit{ and } \exists (x', y') \in M[\![t]\!].\ xWx' \textit{ and } yWy'\}$$
$$M[\![s \circ t]\!] = \{(x, y) \mid \exists z, z'.\ (x, z) \in M[\![s]\!] \textit{ and } zWz' \textit{ and } (z', y) \in M[\![t]\!]\}$$

Likewise, the subset predicate is relative to the wiring:

$$M[\![s \subseteq t]\!] = \forall (x, y) \in M[\![s]\!].\ \exists (x', y') \in M[\![t]\!].\ xWx' \textit{ and } yWy'$$

Note that the meanings of union and intersection are no longer symmetrical in their arguments. The term $s \cup t$ has the type of s. It might instead have the type of t, or, perhaps more naturally, a new type that is the union of the types of s and t. It turns out, however, to be simpler not to introduce new types, and the decision to use s's type is arbitrary. As a result, it is no longer obvious that union is commutative. The terms $s \cup t$ and $t \cup s$ will in general have different meanings; however, any pair (x, y) in one will match a pair (x', y') in the other, with xWx' and yWy', so that we will have

$$t \cup s \subseteq s \cup t$$

and

$$s \cup t \subseteq t \cup s$$

as expected.

We shall call a wired interpretation of a formula f for which f is false a *wired counterexample* of f.

6 *Scopes, Permutations and Adequate Assignment Sets*

In general, a formula will have an unbounded number of interpretations, because one or more of the basic types will be infinite. It is therefore necessary to limit the search for a counterexample to some finite subset of the interpretations. We would like to do this systematically, rather than by just selecting arbitrary interpretations.

A search is conducted within a *scope*, which assigns a natural number to each type name representing a bound on the number of values of the type:

$$\textit{scope}: \textit{type} \rightarrow \textit{Nat}$$

A wired interpretation must map two type names differing only in index to sets of the same cardinality, so any scope must assign the same value to both. We shall therefore not distinguish the conventional scope from the wired scope, since each can be obtained trivially from the other.

For a scope s, we shall say that a type assignment τ is adequate if it maps each type name T to a set whose cardinality is at least $s(T)$. If for any type assignment τ over universe U that is adequate for a scope s there is a variable assignment ρ such that a formula f is false under the interpretation $\langle U, \tau, \rho \rangle$, we shall say that f has a counterexample in the scope s.

For any formula, there is some set $\mathcal{A}$ of possible assignments of relation variables to relation objects. Since the wiring relation effectively permutes these objects, not all assignments will be needed. In enumerating the possible values of a relation variable, out of a set of isomorphic relations only one is required, and this representative may be selected arbitrarily.

A permutation π is a bijection on a finite set X

$$\pi: X \rightarrow X$$

and may be applied pointwise to a relation

$$r \subseteq S \times T$$
$$\pi(r) = \{(\pi(a), \pi(b)) \mid (a, b) \in r\}$$

if it preserves the relation's type, so that $\pi(r) \subseteq S \times T$. To ensure this, we require first that

$$X = S \cup T$$

and second that the permutation π be expressible as the union of separate permutations on the relation's domain and range; that is

$$\pi = \pi_1 \cup \pi_2$$

where

$$\pi_1: S \rightarrow S$$
$$\pi_2: T \rightarrow T.$$

A permutation can be applied pointwise to an assignment ρ under analogous conditions: the domain of the permutation must be the union of the domains and ranges of the relations in ρ, and the permutation must be decomposable into permutations that do not confuse types.The result of applying a legal permutation π to an assignment ρ is

$$\pi\ (\rho) = \{v \mapsto \pi(r) \mid v \mapsto r \in \rho\}$$

Two relations r and s are isomorphic, written $r \approx s$, if there is a permutation π such that

$$\pi(r) = s$$

and two assignments ρ and ρ' are isomorphic, $\rho \approx \rho'$, if there is a permutation π such that $\pi\ (\rho) = \pi\ (\rho')$. Because the types of the individual relations in a wired interpretation are disjoint, a permutation on an assignment can always be decomposed into elementary permutations on individual relations. It follows that two assignments are isomorphic when their constituent relations are pointwise isomorphic.

Given a type assignment τ adequate for some scope s, a set of variable assignments

$$\mathcal{A} = \{\rho_1, \rho_2, \ldots, \rho_n\}$$

is adequate if every variable assignment ρ that respects τ is isomorphic to some assignment in $\mathcal{A}$. Now comes the crucial justification for the notion of wiring:

> *Theorem 1*. Suppose τ is an adequate type assignment over a universe U for a formula f in a scope s. Let $\mathcal{A}$ be an adequate set of variable assignments. Then f has a counterexample in the scope s exactly when it has a wired counterexample $\langle U, \tau, \rho, W\rangle$ for some variable assignment $\rho \in \mathcal{A}$ and wiring W.

In other words, varying the wiring relation achieves the same effect as permuting the relations of the assignment. Instead of checking all assignments, we can pick an adequate set that embodies all assignments of variables to 'shapes' and then enumerate all wirings of these shapes. The benefit of this complication is that the permuting of relations has been localized in the wiring relation, where symmetry can be exploited more simply.

7 *Symmetry in Assignments and Wiring Relation Equivalences*

A permutation π that leaves an assignment ρ invariant

$$\rho = \pi(\rho)$$

is said to be a *symmetry* of ρ. Remember that π acts on the elements of the universe U; it may be applied pointwise to the relations of ρ only when it does not exchange elements of one variable's relation with elements of another's, or elements of a relation's range with its domain. The symmetries of an assignment ρ form a group under functional composition called the *symmetry group of* ρ.

Now suppose we apply the same permutation π to some wiring W (without the constraint, since W is symmetric and π will thus inevitably confuse its domain and range). Over any pair of type sets, W is bijective, so unless π permutes elements in both sets in the same way, it cannot be a symmetry. Nevertheless, even though $\pi W \neq W$, the meaning of the formula under the wiring πW must be the same as the meaning under W:

Theorem 2. If π is a symmetry of ρ then $\langle U, \tau, \rho, W\rangle$ is a wired counterexample of a formula f exactly when $\langle U, \tau, \rho, \pi(W)\rangle$ is.

Consequently, not all wirings need be consided for every assignment. Two wirings W and W' are equivalent for an assignment ρ if there is a symmetry π of ρ such that $W' = \pi(W)$. From each equivalence class of wirings, only one wiring is needed, and it can be picked arbitrarily.

8 Generating Wirings

A wiring relation only matches elements that belong to the same underlying type, so it can be decomposed into a collection of relations, one for each type. Each of these relations wires the elements of the sets $\tau[\![T_1]\!]$, $\tau[\![T_2]\!]$, $\tau[\![T_3]\!]$, etc., corresponding to the underlying type T; between any pair of these sets $\tau[\![T_i]\!]$ and $\tau[\![T_j]\!]$ the relation is bijective. The wiring relation is the union of these individual relations.

Ignoring symmetry then, a full set of wirings can be generated by taking all combinations of individual wirings. These are enumerated as follows. Imagine the sets $\tau[\![T_1]\!]$, $\tau[\![T_2]\!]$, $\tau[\![T_3]\!]$, etc., laid out in a row so that each occupies a vertical column. Take the first element of the first column. This may be matched to any of the elements in the second column, and then to any element of the third, and so on. Each sequence of choices yields a path through the full graph that connects every element in $\tau[\![T_1]\!]$ to every element in $\tau[\![T_2]\!]$, every element in $\tau[\![T_2]\!]$ to every element in $\tau[\![T_3]\!]$ and so on, the path itself being a matching of one element in $\tau[\![T_1]\!]$ to one element in each of the other $\tau[\![T_i]\!]$.

For each of these paths, we now determine the set of paths that start at the second element in the first column but none of which overlap with the path from the first element. Repeating this procedure until no paths remain gives a tree, whose *i*th level represents the choice of path from the *i*th element of the first column. The leaves of the tree then give the wirings, each being obtained by forming a relation from the tuples at each the nodes on the path (this time the *tree* path) from the root to the leaf.

Symmetry allows some of these wirings to be omitted. Theorem 2 tells us that having generated a wiring w we should not subsequently generate a wiring $\pi(w)$ if π is a symmetry of the variable assignment. In the enumeration algorithm, this is avoided by noting that two paths p and p' such that $p' = \pi(p)$ are equivalent, and only one should be generated.

The equivalence of paths is not determined, of course, by attempting to find a suitable permutation. Instead, for each variable assignment, the elements of the universe U are partitioned into equivalence classes. Given any element $u \in U$, and variable assignment ρ, the *orbit* of u is the set of elements that are obtained by applying to it some permutation that is a symmetry of ρ:

$$orbit\ (u, \rho) = \{\pi(u) \mid \exists \pi.\ \pi(\rho) = \rho\}$$

Since the symmetries of ρ form a group, belonging to the same orbit is an equivalence relation and the orbits partition the universe. The symmetries of a variable assignment may thus be represented as a colouring function that maps elements of U to orbits:

$$colour\colon U \rightarrow Orbit$$

Two paths a and b are equivalent when their respective elements belong to the same orbit:

$$equiv\ (\langle a_1, a_2, ..., a_n\rangle, \langle b_1, b_2, ..., b_n\rangle) = \forall i\colon 1..n.\ colour\ (a_i) = colour\ (b_i)$$

The equivalence check ensures that, having built a partial wiring, the algorithm does not generate completions that differ immediately in the choice of the next path selected, when those paths are in fact equivalent. But unnecessary wirings may still be constructed because two paths may be selected in different orders. To rule this out, we pick any ordering on orbits

$$\leq\colon Orbit \leftrightarrow Orbit$$

extend it lexicographically to paths

$$p \leq p' = first\ (p) \leq first\ (p') \lor\ (first\ (p) = first\ (p')\ \land\ rest\ (p) \leq rest\ (p'))$$

and require that paths always be generated in order. Having generated a path p, subsequent choices p' at the same level of the tree must satisfy $p < p'$, and subsequent choices at lower levels of the tree must satisfy $p \leq p'$.

One detail remains: how the orbits are determined. Remember that no permutation exchanges elements of one relation with elements of another. So the orbits of the elements of each relation in the assignment can be determined independently. Our prototype checker constructs the assignments from a fixed repertoire of relation values that are bound to variables in all possible combinations. Each relation value is stored along with its symmetries. Suppose, for example, we have a relation

$$r\colon (S, T)$$

and a type assignment τ with

$\tau[\![S]\!] = \{1, 2, 3\}$
$\tau[\![T]\!] = \{4, 5, 6\}$

Some values r might take, along with their symmetries, are:

relation	*domain symmetry*	*range symmetry*
$\{\}$	$\{\{1, 2, 3\}\}$	$\{\{4, 5, 6\}\}$
$\{(1, 4)\}$	$\{\{1\}, \{2, 3\}\}$	$\{\{4\}, \{5, 6\}\}$
$\{(1, 4), (1, 5)\}$	$\{\{1\}, \{2, 3\}\}$	$\{\{4, 5\}, \{6\}\}$

The second and third columns give the orbit partitionings for $\tau[\![S]\!]$ and $\tau[\![T]\!]$. The colouring function is trivially obtained from these partitionings. Using letters of the alphabet to label orbits

$Orbits = \{A, B, C, ...\}$

these relations might give:

relation	*colouring function*
$\{\}$	$\{1 \mapsto A, 2 \mapsto A, 3 \mapsto A, 4 \mapsto D, 5 \mapsto D, 6 \mapsto D\}$
$\{(1, 4)\}$	$\{1 \mapsto A, 2 \mapsto B, 3 \mapsto B, 4 \mapsto D, 5 \mapsto E, 6 \mapsto E\}$
$\{(1, 4), (1, 5)\}$	$\{1 \mapsto A, 2 \mapsto B, 3 \mapsto B, 4 \mapsto D, 5 \mapsto D, 6 \mapsto E\}$.

9 *Discussion*

Existing checking methods fall into two classes. Those aimed at proving properties of specifications cannot be automatic, since any interesting property of a system whose states are unbounded is undecidable. Theorem provers have been used successfully to verify some substantial designs, but in all cases they require users with the mathematical sophistication to invent lemmas and proof strategies, and a lot of time to spare. For safety critical systems, the investment may be worthwhile, but for routine design work something more economical is called for.

The second class of checking methods is more pragmatically motivated. The specification is 'animated' by calculating the behaviour resulting from specified initial conditions. Such execution typically requires that the specification be written in a constructive form without implicit definitions. The IFAD tool for VDM [ELL94, LL91], for example, can execute explicit function definitions, even when they contain non-deterministic constructs. Z^{--} [Val91] is a subset of Z that requires equalities to be ordered with constructive expressions on the right-hand side so they can be interpreted as assignments. (Z^{--} is not, incidentally, intended as a specification language but as a target for refinement of Z.) Aslantest [DK94] embodies an interesting hybrid approach: the specification can be executed not only over test cases, but also over symbolic initial conditions. The result of a symbolic execution is checked against a specified final condition using a simplifier, which, if always to succeed, must be a theorem prover.

Animating a specification suffers from the same basic problem as testing. When the state space is huge, testing a small number of cases is very unlikely to uncover faults. Counterexamples usually correspond to combinations the designer never considered, and would probably have omitted from a test suite. It is not that testing cannot show the absence of errors; it often fails to show their presence.

The approach described here falls into neither class. Like animation, it is partial, but since the states within the scope are enumerated automatically, no specification of test cases is required. Consequently, the number of cases examined is orders of magnitude greater – 100,000 cases is not a large model checking problem – and more likely to include unanticipated counterexamples. Like theorem proving, the checking method is designed for fully implicit specifications, without demanding a translation into an executable subset.

Counterexample generation lacks the mathematical cachet of proof, but, in engineering terms, it has an advantage. A counterexample has a value quite independent of the specification from which it arose. A designer might be pleased that his design has been formally modelled and proven consistent, but might have doubts about the fidelity of the model. A counterexample, in contrast, can be immediately put to the test; if it reveals an error in the actual design, the model no longer matters.

Symmetry has been used before to reduce state space search, but not, it seems, in the fine-grained fashion proposed here. Clarke, Filkorn and Jha show how global symmetries of the transition relation can be exploited [CFJ94]. The state machine representing a protocol that coordinates the caches of several processors, for example, might be symmetric under the exchanging of one processor for another or one cache line for another. With knowledge of these symmetries (provided to the checker by the user), a quotient machine can be constructed with fewer states than the original machine, each representing a set of states that are equivalent under symmetry. Z specifications do not seem to have global symmetries of this sort. Instead, we exploited the symmetry of one possible relation value a variable may take, in order to reduce the set of states obtained by varying the values of other variables. Note also that we never required any symmetries to be specified by the user: they are inferred by the tool from the symmetries of the underlying relation shapes.

The checking method has been implemented as a prototype in Standard ML of New Jersey. It was built for proof of concept, to demonstrate that symmetry can be used to reduce the search space, and thus provides none of the features a usable tool would require. For example, the decomposition of the wiring relation must be specified by the user, and referenced explictly in the formula. The prototype also runs absymally slowly: it examines about 5,000 states/minute on a Sparc 5. We plan now to construct a usable tool (in C). It will read specifications in a subset of Z and derive the structure of the wiring relation automatically. It will allow the user to experiment with different scopes, constructing, for each, a set of appropriate relation values along with their symmetries. Furthermore, it will offer, for each relation variable, the choice of restricting its values to functions, injections, bijections, etc.

Much research remains to be done. Quantifiers and higher-order relations are not currently handled, nor are interpreted types (such as the integers). Also, a limited form of type inference may be required. If a formula's type declarations are unnecessarily restrictive, the wiring relation will not be decomposed as much as it might be, and the resulting search will be larger. For example, the formula

$$dom\ (p \circ q) \subseteq dom\ p$$

does not require the domain of p and range of q to have the same type. The type declaration

p: (S, T)
q: (T, S)

would give wirings that enumerate all bijections between the domain of p and range of q, even though their relationship is irrelevant. The type declaration

p: (S, T)
q: (T, R)

on the other hand, gives wirings that relate only the range of p and domain of q. This divides the size of the search by the factorial of $scope(S)$.

The fundamental premise of the checking method described here is radical but simple minded: that, in practice, errors in specifications can be detected by systematically exploring even a tiny portion of the state space. Exploiting symmetry can reduce the search dramatically, but it seems unlikely that the exponential nature of the problem will be overcome. This is, perhaps, only a theoretical concern. If constant factors gained by search reduction strategies, by succinct representations and by faster hardware bring the analysis of realistic, industrial designs within reach, asymptotic complexity will surely be irrelevant.

Acknowledgments

Thanks to Somesh Jha and Ed Clarke for explaining their symmetry work to me; to Merrick Furst for some entertaining and useful discussions about graph enumeration; and to the members of the software group for helpful comments on an early presentation of this work.

References

[BC+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Proc. 5th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, June 1990.

[CFJ94] E.M. Clarke, T. Filkorn and S. Jha. Exploiting symmetry in temporal logic model checking. *Fifth International Conference on Computer-Aided Verification*, June 1993.

[CGL92] E.M. Clarke, O. Grumberg and D.E. Long.Model checking and abstraction. *Proc. ACM Symposium of Principles of Programming Languages*, January 1992.

[DK94] Jeffrey Douglas and Richard A. Kemmerer. Aslantest: a symbolic execution tool for testing Aslan formal specifications. *International Symposium on Software Testing and Analysis*, Seattle, August 1994.

[ELL94] Rene Elmstrom, Peter Gorm Larsen and Poul Bogh Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *ACM SIGPLAN Notices*, Vol. 29, No. 9, September 1994.

[GH80] John Guttag and James Horning. Formal specification as a design tool. *7th Symposium on Principles of Programming Languages*, Las Vegas, Nevada, Jan. 1980.

[Jac94] Daniel Jackson. Abstract model checking of infinite specifications. *Proceedings of Formal Methods Europe Conference*, Barcelona, 1994.

[LL91] Peter Gorm Larsen and Poul Bogh Lassen. An executable subset of Meta-IV with loose specification. In S. Prehn, W.J. Toutenel (eds.), *VDM'91: Formal Software Development Methods*, Vol. 1, Lecture Notes in Computer Science 551, Springer-Verlag, 1991.

[MZ94] Peter Mataga and Pamela Zave. Formal specification of telephone features. *Proc. 8th Z User's Meeting*, pp. 29–50, Springer-Verlag, 1994.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, 1989.

[SS93] Gunther Schmidt and Thomas Strohlein. *Relations and Graphs*. EATCS Monographs on Theoretical Computer Science, Springer Verlag, 1993.

[Val91] Samuel H. Valentine. Z^{--}, an executable subset of Z. In J.E. Nicholls (ed.), *Z User Workshop*, York, 1991. Springer-Verlag Workshops in Computing, 1992.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment or administration of its programs on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state or local laws, or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state or local laws, or executive orders.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.